ADAPTIVE MEMORY MANAGEMENT IN A PAGING ENVIRONMENT

Gary Michael Raetz

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ADAPTIVE MEMORY MANAGEMENT

IN A PAGING ENVIRONMENT

by

Gary Michael Raetz

Thesis Advisor: G. L. Barksdale, Jr.

December 1973

Approved for public release; distribution unlimited.

Adaptive Memory Management in a Paging Environment

by

Gary Michael Raetz
Ensign, United States Navy
B.S., Portland State University, 1972

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the
NAVAL POSTGRADUATE SCHOOL
December 1973

ABSTRACT

Adaptive memory management techniques for multiprogramming operating systems are described. Page replacement during execution and initial page assignment are the factors affecting optimal memory usage. Modifications to a time-shared operating system (Michigan Terminal System) that would allow implementation of the Page Fault Frequency Replacement Algorithm are discussed. Additional modifications to this system are suggested that would subordinate job initiation to memory availability.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

The author wishes to express his thanks to his thesis advisor, Assistant Professor Gerald L. Barksdale, Jr., Chairman of the Computer Science Group. Professor Barksdale's time, patience, and advice were essential and much appreciated in the preparation of this paper.

Special thanks must also go to my wife, Mary, whose patience and understanding helped see this thesis through to completion.

I. INTRODUCTION

Adaptive management and allocation of virtual memory for a multiprogrammed computer operating system is the subject of investigation for this thesis. The general problem is to effectively allocate resources to all processes utilizing the system and at the same time minimize the degradation of service to process due to the concurrent running of all other processes. The requirements and characteristics of virtual memory management in a demand-paged system are discussed. A brief discussion of the time-shared multiprogrammed, multiprocessing operating system (Michigan Terminal System, MTS) written for the IBM 360/67 by staff at the University of Michigan is given. A more detailed discussion of the memory management processes employed by MTS is given which establishes the background needed for discussion of adaptive allocation of virtual memory and its proposed implementation.

The concept of a system load factor and factors that are considered in computation of the load factor are discussed[1]. The load factor is employed by the system to make job scheduling decisions. It is suggested that the load factor could also be used in conjuction with some measure of program size to make a decision to allocate or not allocate memory to a program currently requesting to be loaded.

II. VIRTUAL MEMORY MANAGEMENT

In contemporary computing systems, memory is considered as the central resource[22]. This is especially true in a multiprogramming environment since many tasks are sharing memory with the supervisor and other system management procedures. It is for this reason that the storage allocation problem is of considerable interest. The problem is to determine, at each moment of time, how information is to be distributed among the levels of memory[9].

Virtual memory is used to provide a logical address space that is much larger than the physical address space. To accomplish this, a set of addresses different from those provided by physical memory are used and an address translation mechanism is provided. The address translation mechanism translates the program-generated logical addresses into corresponding physical addresses for access by the CPU. The methods of implementing and managing virtual memory are discussed thoroughly in references 4, 9, 18, and 22.

There are three basic techniques used for automatic memory management. They are segmentation, paging, and a combination of both[9]. Segmentation organizes the logical address space into blocks of arbitrary size, each being a linear array of addresses. In theory, each program module is assigned its own segment and each address within a segment is referenced by a two-component address; the segment number and the relative address within the segment. Paging organizes main memory into equal size blocks called page frames. Virtual memory is organized into blocks the same size as page frames; these blocks are called pages. In paging, addresses are also referenced by a two-component address; the page number and the relative address within the page. The third method combines segmentation and paging

into one implementation, thereby accruing the advantages of both[9].

The most widely used paging method implementation is demand paging. Pages are brought into main memory only on occurence of a page fault[22]. Pages are removed from memory on action of the employed replacement policy. The remainder of this paper is devoted to discussion of adaptive replacement policies and implementation of one specific policy, the Page Fault Frequency Replacement Algorithm.

III. THE MICHIGAN TERMINAL SYSTEM

Adaptive management of memory is discussed with reference to one specific operating system, the University of Michigan Multiprogramming Supervisor/Michigan Terminal System (UMMPS/MTS), Version 3.0. This system was created specifically to be the operating system for an IBM System 360 Model 67. The virtual memory allocation scheme is demand-paging, which makes the system suitable for study.

The 360/67 extends the basic System/360 architecture to provide additional capabilities needed in an advanced time sharing system. Multiprogramming, multiprocessing, and multi-access capabilities are incorporated in the Model 67[11]. The Model 67 is the primary computer at the Naval Postgraduate School. Main memory at the NPS installation consists of one million bytes and auxiliary storage consists of one data cell, a 2314 disk unit, and one 2301 drum storage unit[21].

A. THE SUPERVISOR PROGRAM (UMMPS)

The heart of the system is the supervisor program called the University of Michigan Multiprogramming Supervisor, hereafter called UMMPS or the supervisor. UMMPS controls the execution of all other tasks in the system. The supervisor is always resident in processor storage and all addresses accessed by it are real, i.e., the relocation hardware of the Model 67 is not used when UMMPS is executing.

The actual code of the supervisor consists of many subroutines. These subroutines are called in response to hardware interrupts or internal program interrupts. The supervisor appears as an extension of the 360/67 hardware to the tasks it controls[1].

B. MTS

MTS is one of the principle tasks run by the supervisor. MTS is a re-entrant program that is activated once for each user initiated task. User initiated tasks can come from the batch stream or from any one of the supported terminals. MTS is the interface between the user's jobs and the supervisor. Among the many MTS functions are interpreting user commands, file and data manipulation, character conversion, and maintanance of user accounting information. In general, MTS provides the communication between the user and the supervisor/hardware mechanisms[15].

C. HASP

The Houston Automatic Spooling Program (HASP) is another principle task under the control of UMMPS. HASP controls all spooling operations involving card and printed I/O. Another function of HASP is batch task scheduling and initiation. The scheduler monitors the number of active tasks and the load placed on the system by these tasks. It selectively initiates tasks according to a priority which it assigns each incoming batch task. The selection of jobs to initiate depends on the system load and types of jobs waiting initiation. HASP also gives the system operator complete control over all batch tasks, card read/punch, and printing equipment.

D. THE PAGING DRUM PROCESSOR (PDP)

The task that is run by the supervisor to manage virtual memory is called the Paging Drum Processor, hereafter called the PDP. The PDP is a resident system program that controls the reading and writing of pages when the supervisor encounters a page-fault or memory overflow.

Memory is segregated into three distinct levels due to hardware restrictions. The primary memory is core or processor storage, secondary memory is drum storage and tertiary memory is disk storage. The actual hardware to support the paging process are IBM 2365-12 processor storage units, one or more IBM 2301 drum storage units and one or more disk packs of an IBM 2314 disk storage unit.

The PDP is activated when available core storage falls below a critical number of pages or when a currently active task references a logical address currently not available in processor storage. Because the PDP only responds to demands, it will not always be an active task.

The primary unit of information processed by the PDP is the Page Control Block, hereafter called the PCB. A PCB is created by the supervisor for every page that a job requires. PCB's are used by the supervisor to keep track of the exact status of each page. Information contained in each PCB includes page status bits, the page's processor storage address, if valid, the page's virtual memory address, if valid, and the external address, i.e., the drum or disk address if valid. All PCB's are kept in processor storage where the supervisor and PDP can access them[1]. Additional information and the format of the PCB can be found in Appendix A.

The PDP controls the paging process using four queues and five supervisor subroutines[1]. The four queues are the management tools of the PDP. The queues are used to keep track of pages that are to be read in, pages that can be written out to auxiliary storage, pages to free on auxiliary storage, and pages that have been read in. The specific information kept in the queues are the addresses of the PCB's associated with pages in the above situations. The only pages whose PCB's are not on any queue are those pages resident on auxiliary storage. These pages are essentially dormant to the PDP until a task requests that one be read in. At this time the requested page's PCB is placed on a

queue by the supervisor and is eventually read in by the PDP. All pages resident in processor storage have their associated PCB on one of the PDP queues. The reason for this is that all pages in processor storage are available to the PDP for replacement, so the PDP must know their location. Section IV-B gives a more detailed discussion of the paging mechanism and a detailed description of the PDP logic can be found in Appendix B.

IV. MEMORY MANAGEMENT IN MTS

A. ORGANIZATION OF MEMORY, VIRTUAL AND REAL

A general discussion of memory management techniques will be given as well as some discussion of adaptive memory management. The memory management techniques currently used in multiprogrammed paged systems and all discussion of adaptive memory management will be given in terms of what is done in UMMPS. It is therefore necessary to present in some detail the memory organization and management techniques employed in UMMPS.

The characteristics of a demand paged virtual memory system make it necessary for the supervisor program to manage two storage resources, real core and auxiliary storage. The following discussion outlines these two areas in terms of what is done in UMMPS.

Segmentation was mentioned earlier as a means of achieving dynamic relocation. The supervisor uses a modification of the described method called linear segmentation to achieve a large logical-address space[22]. The Model 67, with 24 bit addressing, allows a maximum of 16 segments of virtual storage where each segment consists of 256 pages of 4096 bytes each.

The organization of real core memory, which consists of up to 1 million bytes at the Naval Postgraduate School, is based primarily on four categories of users, each of which requires storage. They are the resident system, system tables and queues created due to other processes, shared paged system programs, and user's tasks.

Segment 0 and 1 are currently reserved for the non-relocatable, non-paged resident system routines[20]. At NPS, these routines currently require about 36 pages of real

core and are always in use, i.e., never available for allocation to any other process.

The supervisor requires storage space to keep information about tasks that it is processing. A section of processor storage is dedicated to the supervisor just for this reason. As a part of the first two segments, it is not paged since it will be used frequently by the system. Examples of information kept in supervisor core are segment tables, page tables, processor queues, PDP queues, and PCB's. All remaining core storage is available to the user on a demand paged basis and is contended for with all other paged processes.

Segment two is used for routines which are paged. Some examples of the shared processes are MTS, KWIC (the system key-word scanner), file building and manipulation routines and other frequently used re-entrant programs. When the system is initially loaded, the shared system programs are loaded in real core pages, as any user program would be loaded. When user programs are loaded into processor storage, pages that are occupied by shared programs may be needed. The shared programs are then removed from processor storage by the PDP to make additional page frames available. The segment two programs then become resident on auxiliary storage and the user task remains in core until its pages are needed by some other task.

There are 13 segments remaining in the virtual address space for user tasks. The present system will only allow nine segments, thus leaving six for user tasks. The six remaining segments are allocated as follows: segment three is allocated to virtual machine programs, segment four is allocated by the system for users task's storage requirements, and segments five through nine are available for allocation by users tasks[20]. A user requesting storage during execution of a program would then be allocated virtual addresses in segment five.

The current hardware configuration at the Naval Postgraduate School allows a total of 256 real core pages if all four memory modules are on line. Given that the system occupies 36 pages and requires some storage for tables and queues, the user has available over 200 real pages for assignment. Virtual memory is much larger as can be seen from the number of segments that can be used. The limit of virtual memory is determined by the capacity of the paging drum and the backup paging disk which is used in case of drum overflow. The capacity of a drum is 900 pages with an expected access time of about eight milliseconds per page. The capacity of the paging disk is 6400 pages with an expected access time of about 80 milliseconds[7].

Virtual memory is made possible by the paging drum and backup paging disk. The primary paging device is the drum since it is nearly 10 times faster than the disk in average page access time. Another feature that makes the drum more suitable to paging is the information storage format.

The IBM 2301 storage drum has 200 addressable tracks which are accessed by the PDP as nine logical tracks or "slots." Each physical track has a capacity of 4 1/2 pages which gives each slot a capacity of 100 pages[10]. Associated with each of the nine slots is a queue of read requests. The queues are serviced in First-in-first-out order. If there are no outstanding read requests for any slot then the PDP will schedule a page to be written on that slot if memory is needed. By scheduling writes on slots with no read requests the PDP makes optimal use of the I/O channel to the drum.

When the drum is full, i.e., 900 virtual pages have been created, the backup paging disk is used. The capacity of the disk is much higher than the drum but the cost in access time makes the disk unacceptable as a primary paging device. The PDP has no provision for optimal I/O on the disk. A channel program is created for each disk page read or written.

While drum storage pages are available the PDP will "page-out" directly from core to the drum. When the drum becomes full, pages from the drum are temporarlily returned to core then paged-out to the disk in a manner which keeps the most used pages on the drum. This aspect of the PDP will be explained in section IV-B.

B. THE PAGING MECHANISM

The PDP and supervisor maintain four queues which keep track of pages that are in various stages of the paging operation. The four queues in actuality are linked lists of PCB's. The four queues and their definitions are as follows:

1. Page In Queue (PIQ) is a list of PCB's for all pages that have been requested from auxiliary storage but which the PDP has not yet started reading.

2. Page In Complete Queue (PICQ) is a list of PCB's for all pages that the PDP has finished reading but has not notified the supervisor. The supervisor periodically checks the PICQ and posts all PCB's from the PICQ to the POQ.

3. Page Out Queue (POQ) is a list of PCB's for all pages that are in processor storage. This list is ordered by it's least recently used (LRU) member and is the list used to get pages for the PDP to write to auxiliary storage when processor storage is needed.

4. Release Page Queue (RPQ) is a list of PCB's for all pages that have been released by the task that owned them. When a task terminates, all pages that were virtual at termination must also be

queued for release. The PDP releases these external addresses immediately upon being made active[1].

These four queues are the communication path between the supervisor and the PDP. The PDP is not always active since the pages in processor storage may satisfy the reference stream of the currently active tasks. When a page fault occurs, i.e., a reference to a logical address not in processor storage, the PCB for that page is linked on the end of the PIQ. The supervisor periodically samples the PIQ and if it finds a PCB it restarts the PDP if it is currently idle. The PDP need not be an inactive task for PCB's on the PIQ to be processed. If when the PDP has finished reading and writing pages on all nine drum positions, there are pages on the PIQ it will process these PCB's as if it were just activated.

Before checking the PIQ the PDP frees all external addresses for PCB's on the RPQ since these pages are no longer needed by any task. It then proceeds to construct channel programs to read a page from each drum position that a page is requested from and gets processor storage for each page. If there is not enough processor storage, the PDP removes pages from processor storage by examining PCB's on the POQ. PCB's whose pages have not been referenced since the last scan of the POQ are candidates for removal. Since the POQ is ordered LRU, the pages that the PDP constructs drum write channel programs for are the unreferenced least recently used pages of processor storage.

If there are any slots that the PDP has not constructed reads for, it will determine the number of pages available in processor storage. If there are less than a preset limit, currently 15, then the PDP will ask the supervisor for pages to write to the drum. The supervisor gives the PDP the least recently used pages' PCBs. The number it

receives depends on the number of empty slots it has in the channel program it is building. The PDP receives less than or equal the number it asks for and constructs the remaining portion of the channel program. If the number of pages in processor storage is greater than the preset limit, the PDP will not write any pages to auxiliary storage.

As well as constructing I/O programs for the drum, the PDP also handles the I/O completion interrupts from the drum at the same time it is constructing reads and writes. Upon receipt of an interrupt, the PDP scans the channel program just completed, looking at the PCB's corresponding to the pages just read or written. For the pages just written to the drum the processor storage page is released since that page now has an external address stored in the PCB. For all pages read from the drum to processor storage the corresponding PCB's are linked on the PICQ. As was mentioned earlier, the supervisor can then link these PCB's on the POQ and restart the task that was in wait due to page-fault.

The manipulation of the POQ is critical since the PCB order on the POQ will in effect be a partial determinant of the paging rate. One of the status bits in each PCB is a reference bit which is turned on each time that the corresponding page is referenced. The supervisor subroutine that manages the POQ and gives PCB's to the PDP for writing performs the ordering of the POQ in the following manner. All PCB's taken from the PICQ are added to the front of the POQ with the reference bit set on. When the POQ is scanned for pages to write out, PCB's are removed from the front of the queue if the reference bit is not on. If a PCB is found with a reference bit on during the scan (i.e., a PCB from the PICQ) it is unlinked from the top of the POQ and linked on the back, and the reference bit is turned off thus accomplishing a least recently used ordered list[1].

If the need for virtual memory exceeds 900 pages then the backup disk is used as a paging device. Since the

expected access time for a page on the disk is about ten times longer than that of the drum, it is very desirable to minimize the number of page-faults that will cause access to the disk. The function of drum storage management is another process performed by the PDP. If the number of available drum pages is less than or equal to 100*(number of drums on line) then the PDP begins moving pages from the drum to the disk. This is called page migration by the University of Michigan. Pages are selected for migration on a least recently used basis here also, where "used" means paged in or paged out[7]. The PDP keeps nine LRU ordered queues, one for each of the drum read positions, on which are linked the PCB's in LRU order. For every drum I/O operation the PDP processes the corresponding I/O complete interrupt. It is at this time that the PCB's referenced during the I/O operation are placed on the back of their corresponding migration queue. At any point in time, the PCB on the front of each migration queue corresponds to the least recently used page on that drum slot and is a candidate for page migration.

Least recently used ordering is used in two processes within the system to help optimize the performance or minimize the effect of paging on the operation of the system. This ordering technique gives the system some information as to the locality of reference of the active tasks whereby the system can make some decision as to which pages will be referenced next and which page frames can be made available for use by other tasks[8]. This replacement technique, which uses information about the currently active tasks, falls within the general category of adaptive memory replacement algorithms.

V. ADAPTIVE MANAGEMENT OF MEMORY

An efficient management and replacement algorithm is of prime importance in a multiprogrammed virtual memory system such as MTS. The memory management technique may make the difference between satisfactory or acceptable response time and unacceptable overhead due to paging. Studies of replacement algorithms have included program behavior independent schemes such as Random and First-in-first-out, to program behavior dependent schemes such as Least Recently Used , Working Set, and Page-Fault Frequency algorithms[5]. The latter can be called adaptive memory management algorithms and have the property of being able to adapt to the dynamically changing memory requirements characteristic of programs. This, in fact, is the definition of an efficient memory replacement algorithm according to Chu and Opderbeck[5].

The advantage of adaptive memory management lies in the capability to dynamically change memory requirements and contents for all active tasks. This permits the system to approach optimal usage of a restricted resource and to minimize degradation due to paging overhead without a priori knowledge of program behavior.

A. CURRENT TECHNIQUES

References 4,5, and 15 give a very good background on the current methods of adaptive memory management. A review of these techniques will be given here, but, if more detail is needed these references are recommended.

The least recently used algorithm was mentioned earlier as being used in MTS as a management scheme. This scheme allows all processes to acquire memory pages without restrictions until an upper bound is reached. The LRU

scheme then comes into play for every page fault that occurs after the upper bound is reached until enough pages are acquired to bring the total pages in use lower than the upper bound. Pages are selected to be paged out based only on when they were last referenced. As Lancaster pointed out, this is strictly a global policy and the paging process of one task can directly affect another, since the page in processor storage referenced farthest back in time may belong to some other task[15].

The Working Set Algorithm, on the other hand, is strictly a local replacement algorithm. The Working Set principle depends on a characteristic of typical execution, that being, that over a short period of time the set of pages referenced is relatively constant. The working set of a given task is a function of the time segment over which the process is monitored. The algorithm is executed at the end of every time segment, removing pages from core that have not been referenced. Optimal performance of this algorithm depends on the time segment length. For programs whose working sets are large, the optimal time segment is different than for programs whose working set is small. The replacement of pages is local to a task's own pages which solves the problem of the LRU algorithm; however, the time segment length problem does not allow optimal use of storage.

Chu and Opderbeck[4,5] have suggested an algorithm that performs page replacement on a local basis and is able to modify its primary decision variable adaptively, thus, overcoming the problems of the LRU and Working Set algorithms. The Page Fault Frequency Algorithm computes the time-between-page-faults (TBPF) and uses this as the key parameter for determining if the faulting task is running with optimal memory allocated. If the TBPF is very short, this is indicative of too few pages allocated to the faulting task. Likewise, if the TBPF is excessively long then pages could be released from the faulting task. When

the TBPF indicates pages are to be released, all pages unreferenced since the last page fault are released thus employing an LRU algorithm for page removal and a global decision parameter. A more detailed description of the PFF algorithm and problems inherent in its implementation can be found in Alexander[1] and Lancaster[15].

B. CURRENT ADAPTIVE MANAGEMENT IN MTS

Processor storage in MTS is managed strictly on a least recently used basis as mentioned earlier, however, the supervisor employs a sceduling algorithm whose policies are partially dictated by paging frequency. This algorithm also has an affect on storage availability depending on the requirements of the currently active tasks.

The least recently used scheme is the only memory management technique used in MTS until the supervisor detects a task that requires extra supervision due to its execessive demands on system resources. At this time, what is called the privileged/non-privileged task mechanism, hereafter called the P/NPTM, becomes an active part of the supervisor and it restrains larger tasks so that they cannot monopolize the system resources.

All tasks are allowed to compete for system resources and hold them until the need for that resource expires. Examples of resources that are competetively acquired are processor time, processor storage, and I/O channels. The scheduling mechanism for the processor does not allow a single task to monopolize the CPU, however, it is possible for a job to acquire more processor storage than should be allowed to permit optimal multiprogramming. To prevent this from occuring, the only characteristic of jobs that the P/NPTM monitors is the amount of processor storage allocated to individual tasks. The P/NPTM simply classifies a task in one of two categories: a neutral task, i.e., a task requiring less than N processor storage pages, and big

tasks, i.e., a task requiring more than N pages, where N is determined by the supervisor based on the total number of processor storage pages available. The upper bound, N, is currently determined to be 16 when four storage modules are available. The total number of pages allocated to big jobs also has an upper bound, M, established by the supervisor. It is currently set to 60.

With the above restrictions, the P/NPTM operates in the following basic manner. If, when a neutral task page faults and it's total processor storage page count is less than N it will remain a neutral task. When any task page-faults and has N processor storage pages then a decision to make it privileged or non-privileged is made. This decision is based on other tasks currently in the system. If there are other privileged tasks (or big tasks), and their total number of processor pages exceeds M, then the faulting task is made non-privileged. A non-privileged task is essentially suspended from all processing until some other privileged task terminates. When a privileged task terminates, the processor storage pages it holds are released, thus lowering the total number of pages allocated to big tasks to less than M. A non-privileged task can then be activated and given privileged status.

When a task attains privileged status, two things are done. The task is allowed to use as many processor storage pages as it requires, and it is given a time slice eight times as long as the normal time slice given neutral tasks. A privileged task remains privileged until it enters a wait state other than page wait or until it uses up it's extended time slice. When either of these events occur, the privileged task is returned to neutral status[1].

The P/NPTM can be classified as a very selective adaptive memory management technique since it manages only those jobs that have high core requirements. It is used to prevent tasks that require much processor storage from monopolizing it. It will also prevent many large tasks from

being active concurrently. If this were not prevented, contention for the limited number of pages in storage would cause a great deal of thrashing[22].

The suspension of a task in the non-privileged state prevents big tasks from acquiring all of processor storage. Suspension will eventually cause more page faults as well. While a task is in the non-privileged state, waiting to gain access to a processor, it's processor storage pages are likely to become the least recently used pages and could possibly be paged out. When status changes from non-privileged, the task is allowed access to a processor and can begin execution. Significant paging overhead could be incurred because the working set for this task is now resident on auxiliary storage. The P/NPTM is beneficial in that it is effective in preventing thrashing, but it will not prevent one large job from monopolizing the system. Once a job becomes privileged, it can monopolize processor storage. The system prevents this by placing all big jobs in a hold queue to be scheduled externally by the operator when resources are available. The inability to adequately schedule and manage all tasks demonstrates the need for an improved management technique.

These management schemes perform satisfactorily but it is suggested by Lancaster[15] that the performance of the system could be improved. The suggested modification is outlined in Reference 11 and entails implementation of the PFF Algorithm. The following section describes considerations important in implementation of the PFF Algorithm in UMMPS.

C. IMPLEMENTATION OF ADAPTIVE MEMORY MANAGEMENT

The simulated performance of the PFF Algorithm has been shown to be better than the best LRU Replacement Algorithm, and is comparable to the Working Set Algorithm[5]. There are two basic requirements to be considered when

implementing the PFF Algorithm. The performance measurement parameter, i.e., the page fault frequency, depends directly upon measurement of the time between page faults for every task. The time must be process (or virtual) time rather than real time to compute the faulting frequency for individual tasks. The 360/67 is equipped with a high resolution timer that updates the timer location every 13 microseconds[13]. The updated timer location is used to keep track of real time and process time for each task. This satisfies the need for an interval timer for the fault frequency calculation. As Chu[5] mentioned in his general discussion of PFF Algorithm implementation, the time of the last page fault must be stored. The reason for this is that if the last page fault occured more than the optimal number of microseconds ago, then some of the task's pages are removed. Storage of the time of last page fault would be quite simple in UMMPS. Each task that the supervisor processed has information about it that must be stored and available to the supervisor. The area assigned each task for this purpose is called a Job Table. Information stored in the Job Table includes the task name, the task's time slice value, register save areas, accounting information, a pointer to the task's PCBs and the number of virtual memory pages the task has. The time between page faults could also be stored in this area.

The other requirement to consider in PFF Algorithm implementation is the data structure used to represent processor storage. Recall that the PFF Algorithm's policy for deallocation of pages is local to the faulting task. The pages removed from processor storage are those unreferenced since the last page fault. These criterion require that a task's pages can be accessed and ordered rapidly and efficiently.

The data structure used by UMMPS to manage removable pages is called the Page Out Queue, as was mentioned earlier. It is simply a linked list of PCB's in least

recently used order. This data structure will not satisfy the PFF requirements since the supervisor would have to search the POQ for PCB's representing the least recently used pages for a faulting task. The time spent in searching would depend on two factors, each of which affects the PCB location on the POQ. The first is how often a task is active and the second is the locality of reference within a task's pages. A more suitable data structure would make access time to a task's least recently used pages independent from these factors.

The data structure suggested for implementation of the PFF Algorithm in UMMPS has the required characteristics. It retains the use of PCB's and requires allocation of one additional block of storage for each task. The additional information block, call it a Task Block, is used as a list head for an individual POQ for each task. There are four essential information fields in each TB, and they are as follows:

1. Task Number: The task number must be included as an identification of the PCB's the TB represents. When searching the modified POQ, the supervisor would base it's search on the task number field to find the list of PCB's associated with the task it is currently processing.

2. Task Page Out Queue (TPOQ): The address of the least recently used page's PCB. This field is essentially a pointer to the LRU ordered list of PCB's for the task represented in field one.

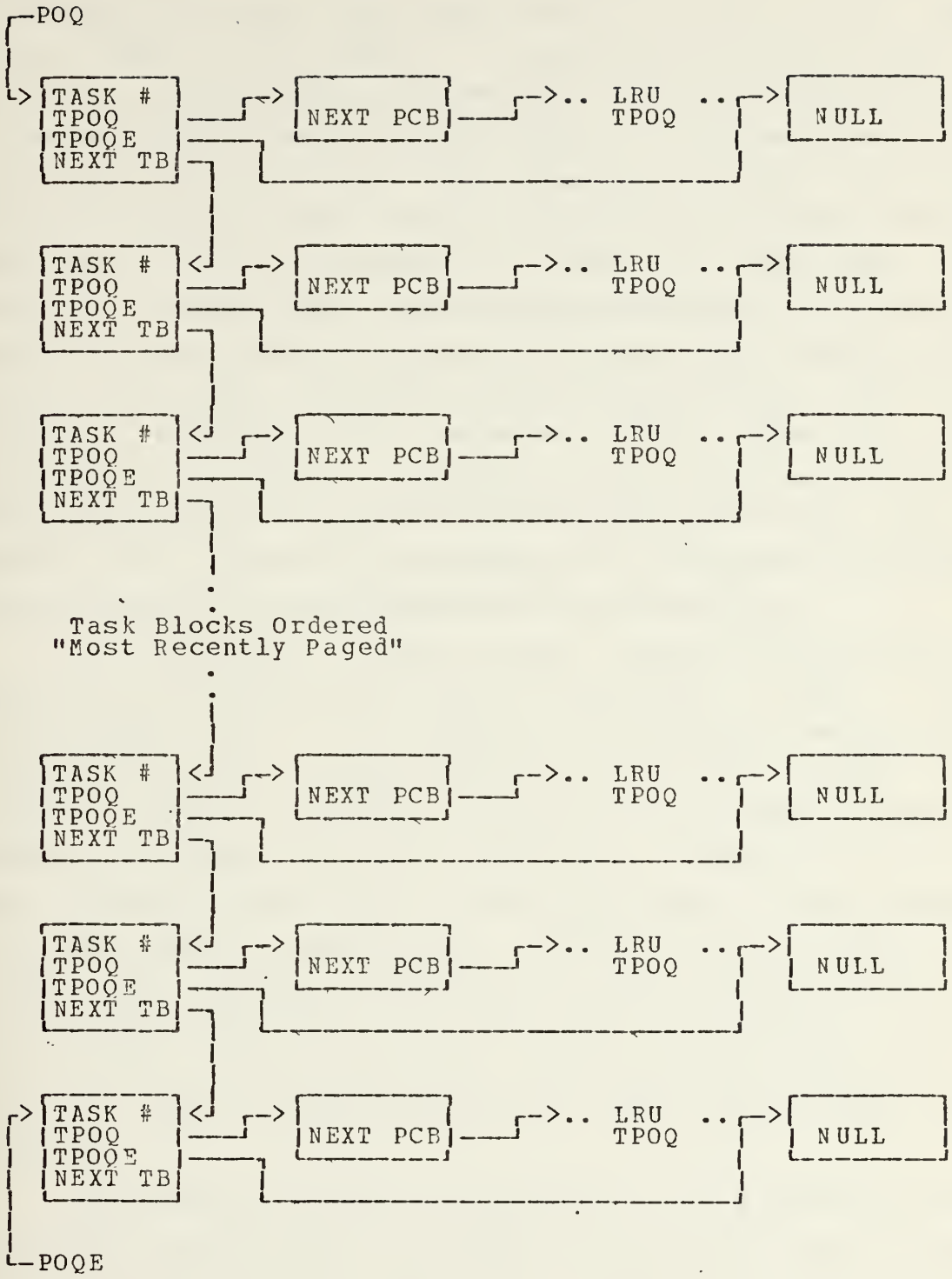
3. Task Page Out Queue End (TPOQE): The address of the last PCB on the above list of PCB's. In ordering the list least recently used, the supervisor must be able to add PCB's to the back of the queue as well as the front.

4. Next Task Block: The address of the Task Block for the next task. This tells the supervisor where the Task Block for the next task is if this Task Block does not match the one it is looking for.

The supervisor now keeps two global variables called POQ and POQE which point to the first and last PCBs on the POQ respectively. With a slight modification to the page creation routine in UMMPS, the variable POQ could point to a Task Block. Within that Task Block would be a pointer to the next Task Block on the POQ. The last Task Block on the queue would have a null value in the Next Task Block field to signify the end of the queue. The supervisor variable POQE would then be set to point to this last Task Block. Figure 1 gives a diagram of the proposed data structure.

Each Task Block is the head of an individual task's POQ. Each of these POQ's would be ordered in the same manner used now, i.e., least recently used. This would be done by preserving the use of the reference and change bits. When a faulting task's pages are to be released, the supervisor then has to scan the list of Task Blocks to find the faulting task's PCB's. When the Task Block is found, the TPOQ field of the Task Block directs the supervisor where to find the task's least recently used PCB.

The suggested data structure conforms to the requirements of the PFF Algorithm's deallocation policy. There will, however, be some additional overhead incurred using the new data structure. The supervisor must search the Task Block list to find the faulting task's PCB's and this takes time. The time required in the search depends on the number of tasks and on the order in which the Task Blocks are kept on the global POQ. It can be seen that if the faulting task's Task Block were at the head of the Task Block list, then the expected search time for that task's



Proposed POQ Data Structure

Figure 1.

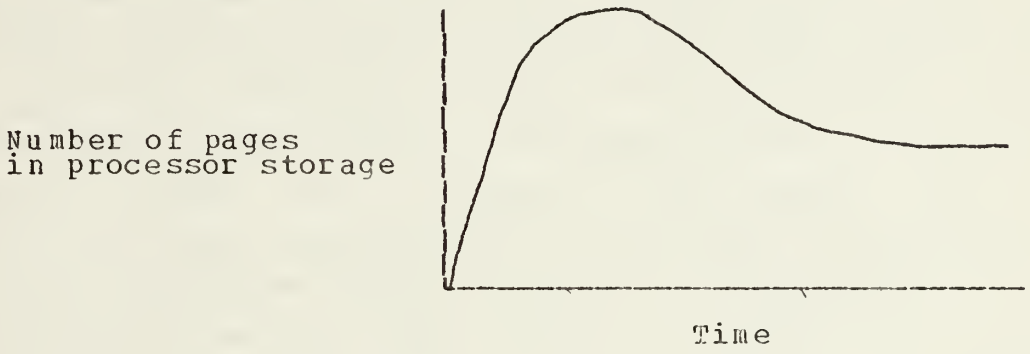
PCB's and pages would be minimized. It has also been observed that when a task page faults, it will often page fault again very soon[1]. Considering these factors, it would be advantageous to dynamically order the global POQ in a "most recently page faulted" order. At the occurence of a page fault for any task, the supervisor should find the faulting task's Task Block and put it at the head of the Task Block list, thus first on the POQ. If the same task faulted several more times during the same time slice and required that pages be removed from processor storage, the search time for its PCBs would be very short. Using this ordering technique, the expected search time for the supervisor to find a faulting task's pages is minimized. The overhead due to the dynamic ordering of the POQ may be offset by the gain in implementation of the PFF Algorithm. This would have to be determined experimentally.

An additional benefit could be realized from use of this data structure when a task terminates. When a task terminates, all it's pages in processor storage and auxiliary paging storage are returned to the system. The system currently releases pages on auxiliary storage by placing the PCB's of pages on auxiliary storage on the RPQ and allowing the PDP to free the pages. The processor storage pages are released by looking in the task's Job Table and finding the location of it's segment table. In the segment table the supervisor finds the location of the page table. The supervisor frees all pages pointed to by the page table and then removes the freed page's PCB's from the POQ.

An alternate method made possible by the suggested data structure would allow the supervisor to free a task's pages by finding that task's Task Block on the POQ. The pages and PCB's could be released by stepping down the TPOQ, releasing pages and PCB's. The global POQ could then be relinked to eliminate that Task Block.

D. LOAD TIME MEMORY MANAGEMENT

The memory requirements for a typical program are usually constant over a short period of execution time. This fact is the basis of the Working Set Replacement Algorithm, as was mentioned earlier. There is a time, however, during the existence of a program in a system, when the processor storage requirements are higher than average. During the loading phase of a program, the memory requirements may greatly exceed the program's working set. This is shown in figure 2.



Program Paging Requirements
Figure 2.

The program loader is a system program that is always resident in supervisor storage. It is invoked each time a user issues an explicit "RUN" or "LOAD" command. The loader is also invoked for every implicit load generated by external references. The loader program's function is to take a program in object deck form, concatenate it with any other object programs that are required, load these object decks into processor storage and transfer execution control to the supervisor. Processor storage pages for a task are initially acquired during the loading of the object program. The loader gets pages for the program until the entire program is loaded.

The loading process could create a sizeable load on a system if the system resources, processor storage in particular, were heavily used at the time the load procedure is started. An extremely heavy paging load would result if a large load procedure were initiated when free processor storage pages are at a minimum. This essentially says that the supervisor should be able to selectively initiate tasks based upon knowledge of the current demands on the system resources.

As was mentioned in the brief description of HASP, the system makes use of an adaptive scheduler to initiate batch tasks. The scheduler assigns an execution priority to each task based upon one parameter supplied by the user, the projected CPU time needed to complete the task. Batch tasks are initiated according to their assigned priority and the current load on the system. The measure of the dynamically changing system load is called a load factor. The load factor is a linear combination of individual system resource factors. The system resource factors are maintained for the supervisor by the JOBS task, and are a measure of resource usage during the last sampling period. Some examples of usage information collected by JOBS are:

1. CPU Activity: This is broken down into two sub-categories; the CPU utilization and the CPU queue length. The utilization is defined as the percent of the available processor time expended during the current sample period. The CPU queue length is the average number of tasks that were running or ready for a processor during the current sample period.

2. Paging Activity: This includes drum page-ins per second and disk page-ins per second.

3. Disk I/O Activity: This gives a measure of the

number of disk channel programs that are being executed per second.

4. Channel Activity: This gives a measure of the load put on the system by starting channel programs on all I/O devices. This includes channel programs started on the disks and drums.

With the above information the supervisor is able to tell which system resources are being used most heavily at any given time. The adaptive scheduler uses this information to determine whether it is appropriate to initiate another batch job. If the projected CPU time of a job is high and the current CPU utilization is low, the scheduler will initiate this job to attempt to increase the utilization of the CPU[19].

There exists one potential problem in this scheduling technique. There could arise a situation where the CPU activity is low but the paging activity is high and processor storage is unavailable. Initiation of a task that requires many pages at load time will increase the paging activity since pages must be acquired for the new task. The extra paging overhead may drastically raise the load factor and degrade the service to all other users. If, however, the supervisor were able to distinguish between pending tasks by the amount of processor storage required and CPU time required then a task that required fewer pages could have been initiated. This would have prevented the drastic rise in the load factor due to unnecessary paging and make better use of the system resources.

E. POSSIBLE IMPLEMENTATION OF ADAPTIVE LOADING

The supervisor has enough information about the current load on the system to make a reasonable decision as to which "RUN" or "LOAD" commands, if any, should be initiated.

There exists a count of available processor storage pages that the supervisor increments and decrements as the paging and loading process proceeds. The free page count in conjunction with the measurements included in the load factor are sufficient to make a decision as to whether the system can afford to give up more free pages and increase the paging rate.

There is one piece of information that the supervisor is currently not able to get until after the loading procedure is finished. The supervisor does not have any measure of processor storage requirements for a program. During the loading process the loader keeps track of how long the load program is, and when the loading is finished the total size of the load program is known. This is too late for the supervisor to find out the size of a load program. A possible means of resolving this problem would be for each loadable object program to have a size signature as well as the name identification currently used. The size signature would be supplied to the object decks by the assemblers, compilers and other programs that create object decks from source code. The size of an object program can be computed by these programs since they are emitting the object code.

With object program length and system load information, the system could make a more optimal decision as to which jobs to initiate. The interpretation of "RUN" and "LOAD" commands could be modified to accommodate the use of the size signature. When one of these commands is given, the location of the object program is also specified. As a preliminary step in the potential load operation, the supervisor could interrogate the signature field of the object program. This would give the supervisor an estimate on the number of pages required to load the program. This estimate would be a lower bound on the pages required and would give the system a decision variable based on core requirements. Using the core requirement estimate in

conjunction with the assigned CPU time priority, the supervisor can make a more appropriate selection of tasks to initiate. If the core requirement estimate exceeds the number of available page frames, the next task in the same priority queue could be considered for initiation. The main advantage of this loading and initiation scheme is that it allows processor scheduling to be subordinate to memory availability. The supervisor can initiate jobs to make optimal use of the CPU and minimize the paging overhead as well. The load factor and utilization factors are the tools used by the system to determine which type of job to initiate and the object deck size and projected CPU time are used to actually select jobs to initiate.

In design of paging systems it is evident that the scheduling function was viewed as the dominant system control function. The allocation procedure was designed as a subordinate function to provide space for jobs as the scheduler dictated[18]. This is evident to some degree in UMMPS. The current UMMPS scheduler observes the system load, memory availability being a factor, but makes no optimal decision as to which task will cause the greatest rise in load factor due to assigning too many pages of processor storage. Over allocation would be allocating more pages to a new task than are available, thus raising the paging rate. The advantage of the suggested adaptive loading technique is that it allows the system to maintain a more constant load on all resources. This is made possible by subordinating the scheduling function to that of allocation which Kuehner and Randell[18] say is the key to improving performance via scheduling.

Efficient processor storage management is a key factor in obtaining optimal performance in a multiprogramming environment. This includes not only management during program execution, but optimal allocation policies at program loading time. Since the processor storage requirements are different and usually higher at load time,

the performance of a system can be adversly affected if resource requirements are not considered in job initiation. This situation exists in the UMMPS environment and a possible solution has been suggested.

VI. CONCLUSIONS

The simulated performance of the Page Fault Frequency Algorithm[4, 15] indicates that it has high potential as a virtual memory management technique. The PFF Algorithm is local to each task in making optimal decisions. The only way to implement this algorithm in UMMPS is to modify the processor storage data structure so that the local policies can be enforced. The given modification to the Page Out Queue will allow the PFF Algorithm to access each task's pages with a minimum of overhead.

PFF, being a local algorithm, cannot optimally manage storage resources on a global basis. For this reason, the processor scheduling algorithm must consider available processor storage and projected storage requirements for a job before scheduling. The current scheduling technique in UMMPS does look at the load on the system before scheduling a job, but it does not take into consideration the potential processor storage requirements of a job. The scheduler also witholds large jobs from the system and requires that they be scheduled externally. This scheduling technique could be improved if scheduling were made subordinate to memory allocation.

The scheduler can schedule jobs to minimize paging and maximize memory use if information about the system and jobs to be scheduled is available. The number of available pages is known and this should be an upper limit on the size of a job to initiate. The information needed is the projected processor storage requirements for jobs to be initiated. The best way to obtain this information is to modify programs that create object decks, such as compilers and assemblers, to identify each object deck with a processor storage requirement signature. With this information, the scheduler can schedule jobs so that the least amount of

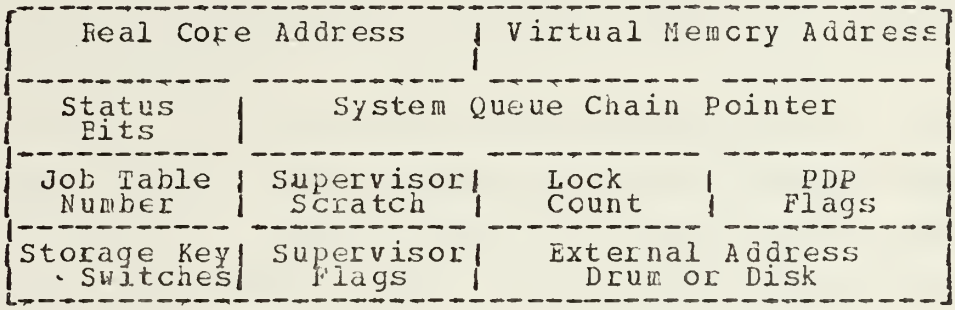
paging is caused, thus approaching optimal management of processor storage on a global basis. This scheduling technique would allow large jobs to be initiated when resources are available.

Supported by a processor scheduling algorithm that can adaptively schedule according to available resources and job requirements, the PFF Algorithm can be implemented in UMMPS. The modifications required to the current system are minor and the potential benefits that may be realized appear to make implementation worth while.

APPENDIX A

FORMAT OF THE PCB IN UMMPS

The Page Control Block is a dedicated block of processor storage created by the supervisor for every page. The PCB requires 16 words of IBM 360/67 processor storage and it's format is as follows:



The word size of the 360/67 is 32 bits, divided into four, eight bit bytes. The size of the fields in the PCB can be computed using the following examples. The Real Core Address field is two bytes and the Status Bits field is one byte in length. The meaning and use of each of these fields is given in the table below.

Real Core Address: If the page represented by a PCB is located in processor storage, then the address of that page is stored in this field. The PDP uses this field to test the page address for validity. If the real core address is not valid or non-existent, this field will contain the value zero. The PDP also uses this address when releasing a real core page. The address only requires 12 of the 16 available bits and is the high order 12 bits of the page's real core address. For example, if the real core address field contained the value 4 hex, the actual

processor storage address of that page is 004000 hex since the 360/67 uses 24 bit addressing. The last 3 bytes, or 12 bits, can represent any address within a page and are not specified within the Real Core Address field.

Virtual Memory Address: When a page is created, it is assigned a virtual memory address. This includes a page number and segment number. The PDP only uses the page number for error diagnostics. The supervisor makes use of this field to manage virtual memory.

Status Bits: The Status Bits field contains four bits which specify the storage key for the second half of the page. Three of the four remaining bits are used for status bits. The status bits are as follows:

Byte Format: 

Storage Key: This field contains a four bit code that is compared with the protection key in the Program Status Word for each reference to storage within the 2048 bytes of the second half of this page. Access to storage is permitted only when the storage key and protection key match, or when the protection key is zero.

Unused: Bit is unused.

Queue Flag: This flag is on if this page is on some system queue. The system queues are the RFQ, POQ, PIQ, and PICQ.

Page Available Flag: This flag is on if this page is available for use. It would be off if this page is not in processor storage.

Release In UNLOCK Flag: On if this page is locked in core and is to be unlocked in supervisor subroutine "UNLOCK" as soon as possible.

System Queue Chain Pointer: The supervisor and the PDP use four queues to keep track of pages as explained earlier. The addresses of the PCB's are used to construct the queues. This 24 bit field is used as a pointer to the next PCB on the same queue as this page. The queue ends are global variables defined in the supervisor and contain the addresses of the first and last PCB on each queue.

Job Table Number: The information in these eight bits tells the supervisor which job table, i.e., which job is using this page. The PDP only uses the job table number for error documentation.

Supervisor Scratch: A temporary work area for the supervisor, unused by the PDP.

Lock Count: Current number of requests for this page to be locked in core. If this count is zero then this page is on the POQ and can be paged out. This information is used by the supervisor.

PDP Flags: This field contains a one bit flag, set by the PDP. It is set when an I/O error occurs in reading a page from auxiliary storage.

Storage Key and Switches: This field contains the storage key for the first half of the page. The function of the storage key was explained in the Status Bits section. There are also three3 flags used by the PDP in this field. The field is defined as follows:

Byte Format: [KKKKNUCS]

Storage Key: Storage key for first half of this page.

Not Used: Bit is unused.

Use Bit: Also called a reference bit, this bit is set each time the page is referenced. It is used by the supervisor in ordering the POQ in a least recently used manner. This bit is reset each time the POQ is scanned for pages to write to auxiliary storage.

Change Bit: This flag is only set when some location within this page is modified. When this page is being considered for writing to auxiliary storage, this bit is checked. If not set, then the processor storage space used by this page can be released without writing the page to drum. This is true since the drum page will be the same as the core copy. If the bit has been set, the page must be written; during the writing process the change bit is reset.

Shared Bit: If this page is shared by more than one task, this bit is on. This flag is only tested by the PDP in an I/O error situation.

Supervisor Flags: The flags in this field tell the supervisor if this page is currently being written to or read from auxiliary storage. There are also two flags used by the Virtual Machine Program which are the virtual use and change bits.

Byte Format: [WNNUCRNN]

Write-Out in Progress: This page is currently being written to the drum if this bit is on.

Not Used: Bits not used.

Use Bit: Virtual Machine use bit.

Change Bit: Virtual Machine change bit.

Read-In in Progress: This page is currently being read from the drum if this bit is on.

External Address: This half-word contains the external address of this page on auxiliary storage. The address can have two forms, a drum address or a disk address. If the page is on drum, the first 3 bits specify which module or drum the page is on. The next 5 bits represent the slot number assigned by the PDP. The last eight bits represent the track number on the drum. If the page is on disk then the first three bits represent the disk pack module number, and the last 13 bits represent the relative page number of this page on the disk pack.

APPENDIX B

LOGIC OF THE PDP

The general logic of the Paging Drum Processor is given. The actual PDP processes are much more complex than presented here, however, a basic understanding of the function and requirements of the PDP can be attained.

When the PDP is started at system initial program load time, it performs initialization of flags, builds queue pointers, locates the actual paging devices, and initializes the paging devices. The initialization process is not included in this write-up, but, the initial values of several PDP internal flags must be known before reading the logic write-up. They are as follows:

1. MIGRATION-IN-PROGRESS: Initially set to false. This flag is true when a page is in the process of being read from drum to core and written from core to disk in the migration process.

2. NEED-MIGRATION: Initially set to false. This flag is true when the number of available drum pages is less than 100 times the number of drums. The PDP will begin a page migration when this flag is true.

3. NEED-WRITE: Initially set to false. This flag is set to true when there are less than a preset number of available page frames in core. When true the PDP will write pages to the drum from the POQ until there are enough available page frames.

4. OPENSLOT: Initially set to false. This flag is set to true when the PDP has an open slot in a Channel Command Word Buffer that could be used for a page migration read or write.

```
STARTPDP
  CHECKRPQ:  If RPQ is empty then go to CHECKPIQ.
      Get PCB at head of RPQ.
      Does PCB have valid External Address?
         No:  Go to NOEXADDR.
         Yes:  If this PCB is migrating, abort migration.
            Remove this PCB from migration queue.
            Free the external disk or drum address.
      NOEXADDR:  Free the supervisor core held by this PCB.
      Go to CHECRPQ.
  CHECKPIQ:  If PIQ is empty then go to ENDCHECK.
      Get PCB at head of PIQ.
      Does PCB have valid External Address?
         Yes:  Go to PUT-ON-LOCAL-PIQ.
         No:  Put PCB on PICQ since page is in core.
            Go to CHECKPIQ.
  PUT-ON-LOCAL-PIQ:  If this PCB is migrating,
      abort migration.
      Remove this PCB from the migration queue.
      Is PCB's page on drum?
         No:  Put PCB on local disk PIQ.  Go to CHECKPIQ.
         Yes:  Set i equal to PCB's slot number from
            external address.
            Put PCB on local PIQ(i).  Go to CHECKPIQ.
  ENDCHECK:  Is MIGRATION-IN-PROGRESS true?
      Yes:  Go to CHECK-FOR-WRITES.
      No:  If there is a drum page address vacated by a
         migrated page then free this external address.
         Is page migration needed, i.e., is there less than
         (100*(number of drums)) free pages?
            Yes:  Set NEED-MIGRATION to true.
               Go to CHECK-FOR-WRITES.
            No:  Release the core page used for migration,
```

if one exists.

CHECK-FOR-WRITES: Is core too full, i.e., do pages need to be written out to free page frames?

Yes: Set NEED-WRITE to true.

STARTLOOP: If there are less than two PCBs on the local PIQ or (NEED-MIGRATION and NEED-WRITE) are false then go to RLABEL.

If there is no available Channel Command Word Buffer then go to RLABEL.

Get a CCW Buffer. If the buffer is not initialized, then initialize the buffer.

Set i equal to 1.

Set EMPTY-CCW-SLOTS to zero.

Set OPENSLOT to false.

CCW1: If local PIQ for slot(i) is empty then go to CCW2.

Get PCB at head of local PIQ for slot(i).

Get a real core page frame if possible. If not go to CCW2.

Put Real Core Address of page in PCB.

Build read CCW if word i of CCW Buffer for PCB on slot(i). Include address of real core page just acquired.

CCW3 Add one to i. If i≤9 go to CCW1 else go to BUILD-MIG-CCW.

CCW2: Set OPENSLOT to true since slot(i) is available for page migration use. If drum pages available on slot(i) is not zero then increment EMPTY-CCW-SLOTS by one. Go to CCW3.

BUILD-MIG-CCW: Is NEEDMIG true?

No: Go to MAKE-WRITE-CCW.

Yes: Is OPENSLOT true?

No: Go to STARTIO since no slots to write to.

Yes: Find an open slot. Set i equal to open slot number.

MIG-GET-PCB: Get PCB at head of migration queue for slot(i). It represents the oldest page on

slot(i). If Real Core Address for page is valid release the drum address and go to MIG-GET-PCE.
Is there a real core page to read migration page into?

No: Get a real core page frame if possible.
If no core available go to MAKE-WRITE-CCW.

HAVECORE: Remove PCB from migration queue for slot(i) and decrement EMPTY-CCW-SLOTS by one.
Construct CCW for drum read in CCW Buffer word i.
Set NEEDMIG to false.

MAKE-WRITE-CCW: If EMPTY-CCW-SLOTS is zero go to STARTIO
Set NEED-WRITE to false.
Are there any pages to write that previoulsy failed to be written?

Yes: Set PDP Page Out Queue equal to write-retry queue. Get PCB at head of PDP POQ. Go tc POQ-WRITE.

GET-POQ: No: If pages need not be written to free core page frames go to STARTIO else set PDP Page Out Queue to same number of PCBs on supervisor POQ as there are EMPTY-CCW-SLOTS. Get PCB at head of PDP POQ.

POQ-WRITE: Is external address of page valid?
No: Go to MUST-WRITE.
Yes: If the change bit is set, go to MUST-WRITE.
If External Address is on disk, go to MUST-WRITE.
Free the Real Core Address for the page.
If page was reclaimed, go to PAGE-RECLAIMED.
If page does not exist in core go tc PAGE-RELEASED.
Go to TRY-AGAIN.

PAGE-RECLAIMED: Page in core was reclaimed by user.
Is this page migrating? Yes: Abort migration.
Remove the PCB from the migration queue.
Free the external drum or disk address.
Go to TRY-AGAIN.

PAGE-RELEASED: Page does not exist, task terminated.

Is this page migrating? Yes: Abort migration.
Remove this PCB from migration queue.
Free the supervisor core for this PCB.
Free the drum or disk External Address.
Go to TRY-AGAIN.

MUST-WRITE: Set i to an empty slot number.
If page for the PCB is migrating, abort migration.
Remove PCB from the migration queue.
If the PCB has old External Address, delete it in the PCB and free the drum or disk address.
Get an external address on slot(i) if possible. If unsuccessful go to MUST-WRITE.
If page is to be written to disk go to MAKE-DISKW-CCW.
Construct write CCW in word i of CCW Buffer for this PCB.

TRY-AGAIN: Get next PCB on PDP Page Out Queue,
if queue not empty, and go to POQ-WRITE.
If PDP POQ is empty and there are still available slots for writes, go to GET-POQ, else go to STARTIO.

MAKE-DISKW-CCW: Construct write CCW in disk CCW Buffer for this PCB. Go to STARTIO.

STARTIO: If there is no CCW Buffer to use go to RLABEL.
Start I/O with filled CCW Buffer. On I/O completion go to POST-I/O-COMPLETE.

POST-I/O-COMPLETE: For all PCBs whose page was written, free the real core page unless the following occur;
If the core page was reclaimed then free the external drum or disk address.
If the page was released by the task, then free the external drum or disk address and free the core PCB.
For all pages that were read, put the associated PCBs on the local PDP PICQ.
Put all I/O completed PCBs on the tail of the migration queue for the appropriate slot.
Free the CCW Buffer.

If the PDP was waiting for a CCW Buffer, go to STARTPDP, else go to RLABEL.

RLABEL: If there is work for the PDP to do, go to STARTPDP. This branch would be taken if there is I/O to be started, i.e., an unexecuted CCW Buffer. If not, chain all PCBs on the PDP PICQ together via the System Queue Chain Pointer.

CHECKPICQ: If the supervisor PICQ is not empty, then request that PDP be put on bottom of CPU queue. When reactivated go to CHECKPICQ.
If the supervisor PICQ is empty, chain the PDP PICQ to the supervisor PICQ, emptying the PDP PICQ.
If there has been no work posted for the PDP to do then stop, else go to STARTPDP.
After the PDP has stopped, the supervisor will restart it when PCBs are put on the RPQ and PIQ.

BIBLIOGRAPHY

1. Alexander, Michael T., Timesharing Supervisor Programs, The University of Michigan Computing Center May 1969, Rev. May 1970.

2. Alexander, Michael T., "Organization and Features of the Michigan Terminal System," AFIPS Conference Proceedings, v. 40, P. 585-591, 1972.

3. Arden, B. and Boettner, D., " Measurement and Performance of a Multiprogramming System," Second ACM Symposium on Operating Systems Principles, Princeton, N.J., October 20-22, 1969, P. 130-146.

4. Belady, L.A., "A Study of Replacement Algorithms For A Virtual Storage Computer," IBM Systems Journal, v. 5, No. 2, 1966.

5. Chu, W.W. and Opderbeck, H., "The Page Fault Frequency Algorithm," AFIPS Conference Proceedings, v. 41, AFIPS Press, Montvale, N.J..

6. Chu, W.W. and Opderbeck, H., Performance of Replacement Algorithms With Different Page Sizes, Computer Science Dept. University Of California, Los Angeles, California.

7. Computing Center Newsletter, University of Michigan Computing Center, v. 2, No. 14, October 9 1972.

8. Denning, P.J., "The Working Set Model For Program Behavior," Comm. ACM, v. 11, No. 5, P.323-333, May 1968.

9. Denning, P.J., " Virtual Memory," Computing Surveys, v. 2, No. 3, September 1970.

10. Fuller, S.H., "Performance of an I/O Channel With Multiple Paging Drums," SIGME Symposium on Measurement and Evaluation Proceedings, Association For Computing Machinery, 1973.

11. Gibson, Charles T., "Time-sharing in the IBM/360: Model 67," AFIPS Conference Proceedings, v. 28, P. 61-78, 1966.

12. Hinson, E.F., A Comparitive Study Of The Michigan Terminal System (MTS) With Other Timesharing Systems For The IBM 360/67 Computer, Masters Thesis, Naval Postgraduate School, Monterey California, 1971.

13. IBM System/360 Priciples of Operation, Eigth Edition, International Business Machines Corporation, File No. S360-01, Form A22-6821-7, September 1968.

14. Irwin, D.J. and Thoringer, J.M., "An Adaptive Replacement Algorithm For Paged Memory Computer Systems," IEEE Transactions On Computers, v. C-21, No. 10, Oct. 1972.

15. Lancaster, A.E., Implementation Of The Page Fault Frequency Replacement Algorithm, Masters Thesis, Naval Postgraduate School, Monterey California, 1973.

16. Parmlee, P.P. and others, "Virtual Storage And Virtual Machine Concepts ," IBM Systems Journal, v. 12, No. 2, 1972.

17. Randell, B. and Kuehner, C.J., "Dynamic Storage Allocation Systems," Comm ACM, v. 11, No. 5, P. 297-306, May 1968.

18. Randell, B. and Keuhner, C.J., "Demand Paging In Perspective," AFIPS Conference Proceedings, v. 33, Part II, P. 1011-1018 , 1968.

19. University of Michigan, MTS Volume 1: MTS and the Computing Center, P. 112, 3d ed., January 1973.

20. University of Michigan, MTS Volume 5: System Services, 3d ed., December 1971.

21. Users Manual, First Edition (with Change 20), William R. Church Computer Center, Naval Postgraduate School, Monterey, California, P. 1-7 Figure 1.3, March 1970.

22. Watson, Richard W., Timesharing System Design Concepts, P. 135-184, McGraw-Hill, Inc., 1970.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Professor G. L. Barksdale, Code 72 Naval Postgraduate School Monterey, California 93940	1
4.	Ens Gary Michael Raetz, USN Computer Science Group Naval Postgraduate School Monterey, California 93940	1
5.	Chairman, Computer Science Group, Code 72 Naval Postgraduate School Monterey, California 93940	1
6.	LTJG T. G. Price, Code 52Pg Naval Postgraduate School Monterey, California 93940	1

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Adaptive Memory Management in a Paging Environment		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis (December 1973) 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Gary Michael Raetz, ENS, USN		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE December 1973 13. NUMBER OF PAGES 54
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Virtual memory
Time-sharing systems
Paged memory
Michigan Terminal System
MTS
Adaptive Memory Management
Replacement Algorithms
Operating systems

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Adaptive memory management techniques for multiprogramming operating systems are described. Page replacement during execution and initial page assignment are the factors affecting optimal memory usage. Modifications to a time-shared operating system (Michigan Terminal System) that would allow implementation of the Page Fault Frequency Replacement Algorithm are discussed. Additional modifications to this system are suggested that would subordinate job initiation to memory availability.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |
(Page 1)

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*